NASA Contractor Report 189613
ICASE Report No. 92–8

ICASE

EXECUTION MODELS FOR MAPPING PROGRAMS ONTO DISTRIBUTED MEMORY PARALLEL COMPUTERS

Alan Sussman

Contract No. NAS1–18605
March 1992

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665–5225

Operated by the Universities Space Research Association

NASA
National Aeronautics and
Space Administration

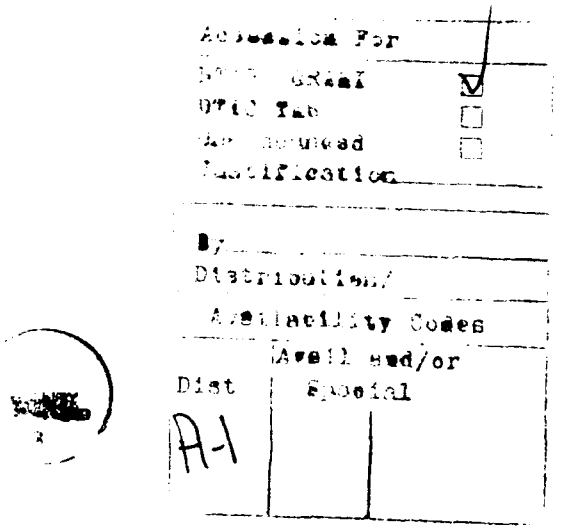
Langley Research Center
Hampton, Virginia 23665-5225

EXECUTION MODELS FOR MAPPING PROGRAMS ONTO DISTRIBUTED MEMORY PARALLEL COMPUTERS

Alan Sussman[1]
Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA 23665
als@icase.edu

This paper addresses the problem of exploiting the parallelism available in a program to efficiently employ the resources of the target machine, in the context of building a mapping compiler for a distributed memory parallel machine. The paper describes using *execution models* to drive the process of mapping a program in the most efficient way onto a particular machine.

Through analysis of the execution models for several mapping techniques for one class of programs, we show that the selection of the best technique for a particular program instance can make a significant difference in performance. On the other hand, the results of benchmarks from an implementation of a mapping compiler show that our execution models are accurate enough to select the best mapping technique for a given program.



[1]This work was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18605 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE). While the author was a student in the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, the research was sponsored by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title *Research on Parallel Computing* issued by DARPA/CMO under Contract MDA972-90-C-0035, ARPA order No. 7330.

1 Introduction

Programming a distributed memory parallel computer is a notoriously difficult task. One approach to alleviating those difficulties is to provide tools that automatically map a single-threaded program description into the multiple programs that will run on each processor of the parallel machine. Such tools must also generate the communication between processors that is necessary to correctly execute the parallel version of the program. Many different methods to map programs onto distributed memory parallel machines have been described, with each method tailored to the requirements of both the program and the machine. In addition, some tools have been built that apply a single mapping strategy to each input program, to generate a parallel program for a given machine. However, relatively little work has been done on deciding what is the *best* mapping method, of those that are applicable, for a particular program on a particular machine. This paper promotes *execution modeling* as an effective method for guiding the process of automatically mapping a program onto a distributed memory parallel computer.

A mapping compiler for a distributed memory parallel processor can solve many of the problems that exist in writing efficient programs for such a machine. This paper presents a method for automatically mapping programs onto the distributed memory parallel machine that has two components: mapping techniques and execution models. The compiler writer selects a set of efficient structured techniques for mapping programs onto the target machine and then builds execution models to predict the run-time behavior of programs mapped using those techniques. This task is analogous to building the code generation part of a compiler for a sequential machine, where the compiler writer must find criteria for selecting the best code sequences to perform a particular computation. The execution models are used by the mapping compiler to guide the selection of the most efficient method for mapping a particular program onto the target machine. The results from experiments with a mapping compiler for Sisal on Warp show that this approach is effective for automatically mapping programs onto a real distributed memory parallel machine. By accurately modeling the run-time behavior of mapped programs, a mapping compiler can provide both a wide range of mapping techniques, and the ability to select the most efficient technique(s) for a particular program.

The process of transforming a single-threaded input program into executable programs for each processor in the distributed memory parallel machine has several stages. Since this paper does not address the problem of finding the parallelism available in a program, we assume that the mapping compiler takes, as input, a program that has already been transformed (or originally written) to expose parallelism. The mapping compiler uses the execution models to select from among a set of mapping techniques that can be applied to the input program, and produces a separate program for each processor in the distributed memory parallel machine. We call these separate programs *cell programs*, and assume that cell programs are in the form of a sequential high-level language program (e.g. Fortran, C, etc.), aug-

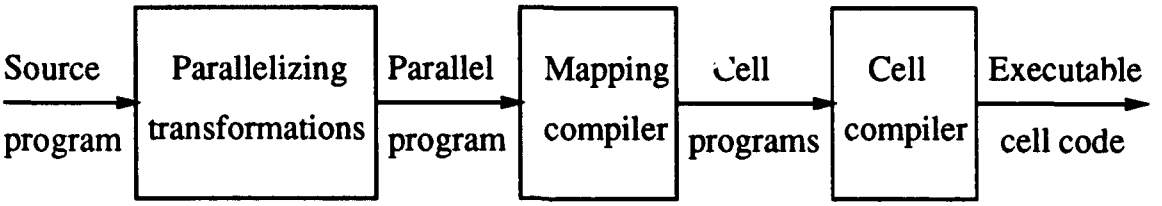


Figure 1: Compiling a program onto a distributed memory parallel machine

mented with send and receive primitives for communication between processors. A compiler for the cell programs, called a *cell compiler*, generates executable code for each processor. Utilizing the cell compiler in this manner allows the mapping compiler to benefit from any optimization techniques developed specifically for producing code for the individual processors in the machine, and allows the mapping compiler to ignore many of the low-level details of the individual processors. The entire compilation process is depicted in Figure 1.

The approach to mapping programs onto the parallel machine is compiler-oriented, in that the ultimate application of the execution models and mapping techniques is a compiler that efficiently maps a single program onto the multiple processors of a distributed memory parallel machine. Therefore, the set of mapping techniques considered is constrained to those that are well-structured, so they can be applied automatically by a compiler. The execution models for the machine and for the mapping techniques are limited to being inexpensive to evaluate, so they can be used by a compiler to guide the mapping process.

1.1 Analysis

Detailed analyses of the execution time behavior of one class of programs, mapped in multiple ways onto a linear processor array, are presented in the paper. Analyzing the execution models for different mapping techniques that can be applied to the same program allows the determination of the values of the program parameters for which each mapping technique performs best. The program parameters include the size of the data set(s) and the amount of computation in the body of a loop.

If we vary the values of program (e.g. data set size) and machine parameters (e.g. number of processors), the execution models can provide information about the sensitivity of the execution time of a program, mapped in a particular way, to such changes. In comparing two mapping techniques that can be applied to the same program, a comparative analysis can determine the conditions under which one of the techniques is preferred, because it will perform better on the machine. Significant differences in the performance of code generated using competing techniques are possible, as will be shown in the analyses in Section 3. Such performance differences indicate that the ability to choose from more than one mapping tech-

nique is a significant enhancement to the capabilities of a mapping compiler.

1.2 Evaluation

While the general model of execution presented in this paper applies to mapping any program onto any distributed memory parallel machine, using a structured mapping technique, uncertainty in the values of various parameters of the program and machine can lead to great difficulties in predicting the execution time of real programs on a real machine. The values of many program and machine parameters can be difficult to obtain. For example, it can be difficult to estimate the execution time of the code assigned to each processor, or to predict the latency of messages propagating through the communication paths in the machine. We therefore present the results of implementing a set of mapping techniques and their associated execution models for Sisal [12] programs on the Warp systolic array machine [2], to show that uncertainty in the parameter values does not significantly affect the accuracy of the models. In particular, the models are accurate enough that the mapping compiler can select the best technique from among those applicable, for a given program.

2 A General Model of Execution

A general model for predicting the execution time behavior of a program on a distributed memory parallel machine must model both the behavior of the individual processors in the machine and the behavior of communication between processors. The modeling method presented in this paper is targeted at structured mapping techniques, meaning those that can be applied automatically by a mapping compiler. To build an execution model for a class of programs mapped with a particular technique onto a given machine, the parameters of both the machine (e.g. number of processors, interprocessor connection topology, etc.) and the program (e.g. data set size, data access patterns, etc.) must be considered. In addition, the mapping technique provides information on the allocation of data and computation to the various processors. All three of these components must be considered in building an accurate model of execution for the mapped program on the distributed memory parallel machine.

2.1 A general framework

Since we are not imposing restrictions on either the programs to be modeled or the mapping techniques to be applied, the only fixed point on which to base a general model of execution is the parallel machine. If we examine the behavior of a program executed on a distributed memory parallel machine from the point of view of a single processor in the machine, there are only three possible activities that

the processor can be performing at any given time: local (sequential) computation, communication with another processor, or waiting (either for data to be received from or sent to another processor). In some machines, a processor can perform computation and communication at the same time, so a general model must also account for any overlap.

In a distributed memory parallel machine with P processors, there are four parts to the execution model of a mapped program for processor i, $\forall 1 \leq i \leq P$:

- SEQ_i - local, sequential computation
- CL_i - communication, to receive and/or send data
- SD_i - synchronization delay; waiting either for data to be received or sent
- OL_i - overlap between computation (SEQ_i) and communication (CL_i)

We will use this terminology throughout the paper to describe the various components of the execution models for mapped programs.

SEQ_i, CL_i, SD_i and OL_i completely characterize the execution time behavior of the mapped program on processor i in the distributed memory parallel machine, and the total execution time for the mapped program on the processor is given by (1).

$$T_i = SEQ_i + CL_i + SD_i - OL_i \tag{1}$$

The execution time for the mapped program on the entire distributed memory parallel machine with P processors is then determined by the processor with the greatest total execution time, as shown in (2).

$$T = \max_{1 \leq i \leq P} T_i \tag{2}$$

The formulas in (1) and (2) provide a way to model the execution time of any program mapped onto any distributed memory parallel machine, provided that the four parts of the model (SEQ_i, CL_i, SD_i and OL_i) can be characterized for each processor in the machine. In practice, the structure of a mapping technique can be exploited, so the behavior of all processors, for each of the four components, does not have to be characterized in detail. We are only interested in structured mapping techniques, because those are the ones that can be applied automatically by a mapping compiler. We now discuss the four components of the general execution model in more detail.

2.1.1 Local, sequential computation on a processor

SEQ_i is the local computation component of the general execution model. SEQ_i models the execution time of the computation that a mapping technique for a program assigns to a processor. In addition, SEQ_i includes the local memory

management overhead introduced by some mapping techniques (to optimize the use of limited local memory). Determining SEQ_i is a completely local analysis, because it only requires analyzing the behavior of the computation assigned locally to a processor, and not the computation assigned to other processors.

2.1.2 Communication between processors

CL_i models the communication time for messages received and sent by a processor. CL_i includes all the execution time for a processor to receive all the data required from other processors, and all the time to send data to other processors. CL_i also includes the time for a processor to pass through data that is destined for other processors. In general, for a structured mapping technique, all the messages that will be transmitted for a given program are known at mapping time, so CL_i can be determined. Computing CL_i requires only local analysis of the messages received and sent by a processor.

2.1.3 Overlap between computation and communication

OL_i models the overlap between computation (SEQ_i) and communication (CL_i) on a processor. If a processor can perform non-blocking sends and receives, or can perform both computation and communication in the same instruction (e.g. the Warp systolic array machine [2]), then an accurate execution model must account for potential overlap. In a machine with processors that can perform computation and communication at the same time, designing mapping techniques that take advantage of that capability is crucial in obtaining the best performance from the machine. Modeling the overlap for a mapping technique on a distributed memory parallel machine allows the designer of a mapping technique to determine how much of the communication overhead required to implement the technique can be hidden by useful computation. Determining OL_i requires only local analysis of the computation and communication assigned to a processor by a mapping technique.

2.1.4 Synchronization delay

SD_i models the time a processor spends blocked waiting for data either to be received or sent. A processor can block to receive data for various reasons, for example, when receiving input data from an external device (e.g. disk, network, etc.), or when receiving data produced by another processor. On the other hand, a processor can block when sending data to another processor if the queue for the data is full on the receiving processor. Such implicit synchronization is usually implemented in the communications hardware of the distributed memory parallel machine, but deciding when synchronization delay will occur for a message can be determined from the structure of the mapping technique. In particular, the pattern of communication required by a mapping technique for a given program can be used to determine the synchronization delays for the various processors taking part in

the communication. Determining SD_i requires global analysis of the communication and computation patterns a mapping technique generates for a program. The delays a processor experiences during mapped program execution are a function of all the messages each processor receives and sends, so the total synchronization delay depends on the execution-time behavior of the mapped program on all processors. Fortunately, all the structured mapping techniques we have investigated produce message patterns with a discernible structure that can be exploited, so the synchronization delays for the various processors can be determined.

2.2 Related work

Several efforts have been made to model general program execution on an MIMD distributed memory parallel computer. Cytron [7] describes models for choosing the number of processors needed to minimize execution time (or maximize processor utilization) of parallel (doall) loops and reduction operations on ring and tree-based processor networks. Chen et al. [6] describe a general model for predicting program performance, in the context of choosing optimization strategies for mapping Crystal functional programs onto hypercube multicomputers.

Work has also been done on modeling the run-time performance of programs executed on shared memory multiprocessors. Sarkar [14, 15, 16] uses execution profile information to approximate the time to execute a node in a dataflow graph, derived from an applicative program, on a multiprocessor . Such information must be generated by running the program, so his approach cannot solve the problem of deciding *how* to map a program without actually producing the mapped program. The PTRAN analyzer [1, 5] can generate estimated execution times for Fortran programs, including estimates for parallelizing loops on shared-memory multiprocessors. The estimates characterize the performance of a sequential Fortran program and estimate the speedup obtainable from parallel execution of the program. Polychronopoulos et al. [13] describe a static program analyzer for Parafrase-2 that obtains compile-time estimates of execution time using a general, parameterized model for program execution on a shared memory multiprocessor. To model sequential execution time, the analyzer uses the longest path through the program (over all branches) to obtain an estimate, but can also use the average time over all paths for a more realistic sequential model. Atapattu and Gannon [3] describe performance models to help programmers find bottlenecks in parallel programs. The models use the assembly code for each processor, generated by a parallelizing compiler, to model execution-time behavior.

Another class of work on modeling program execution concentrates on particular types of programs and/or structured mapping techniques. Hudak and Abraham [10] describe models for data partitioning a sequentially iterated parallel loop onto a shared memory parallel machine. King et al. [11] describe models for pipelined data parallel algorithms, using timed Petri nets, on linear processor arrays and two-dimensional processor meshes. Balasundaram et al. [4] describe a perfor-

mance estimator to select a data decomposition strategy. A set of routines is used to 'train' the estimator, to determine both how long various operations take to execute on one processor, and how long various communication patterns take to complete on the machine. The estimator only works on programs implemented with the loosely synchronous programming model [8].

3 Analysis of Mapping Techniques

The general execution model presented in Section 2 can be applied to various methods for mapping the same program onto a distributed memory parallel machine. For a program that can be mapped in multiple ways, the models can be used to determine which method produces the best performance on the machine, for a particular set of values of program and machine parameters. In this section, we apply the general execution model to analyze, in detail, the execution time behavior of a linear processor array on a parallel loop program, mapped both by data partitioning and by pipelining the body of the parallel loop. Analysis of mapping techniques for a more complex program, a sequentially iterated parallel loop [10], are presented in the author's dissertation [18]. We develop execution models for each mapping technique under various assumptions about the structure of the code generated by the mapping technique for the processors in the parallel machine, and then apply those execution models to compare the performance of the different mapping techniques for various values of the parameters of the program.

In analyzing the execution time behavior of programs on the parallel machine, all execution time behavior is presented in terms of arbitrary time units. The time units can represent processor cycles, processor instructions counts, or any other measure of processor execution time. For a real machine, the units should correspond directly to processor execution time (in seconds). Throughout the analysis, we will refer to these time units as an *execution cost* to perform an activity (computation or communication) on a processor.

In the following analyses, we assume that the communication cost on the machine can be modeled as one cycle per data item sent or received. The assumption is accurate for a machine with programmed communication, such as the Warp systolic array machine, but may not be accurate for message-passing machines (where a message startup latency may occur). The assumption can be modified if there is a better communication model for the machine.

In analyzing the parallel loop program, we only consider the mapped program behavior for program input data set(s), and not for output data set(s). For this program and its associated mapping techniques, output data are handled analogously to input data (instead of distributing data, collect data), and the execution costs involved are exactly the same.

The linear processor array is viewed as an attached processor, with a host machine supplying input data and collecting output data. As shown in Figure 2,

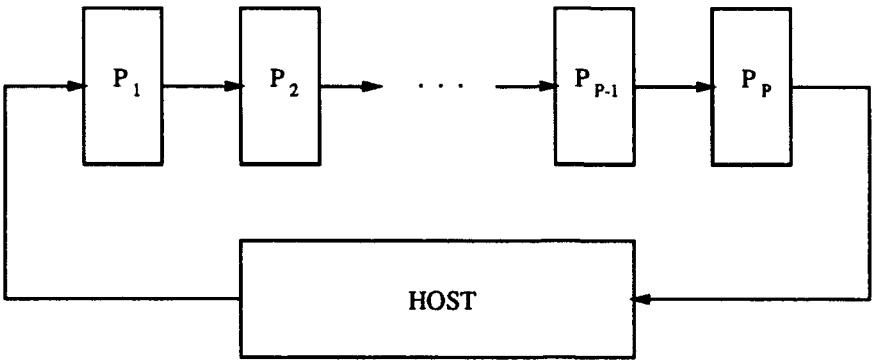


Figure 2: A linear processor array, attached to a host machine

```
forall i in 1, N
  out[i] := f(in[i-x], in[i-x+1], ... , in[i], ... ,
                                          in[i+y-1], in[i+y])
end
```

Figure 3: Parallel loop with neighborhood computation

each processor in the array can receive data from the processor to its left and send data to the processor to its right, except for the first (P_1) and last (P_P) processors in the array. The first processor in the array receives input data from the host machine and the last processor sends output data to the host. The number of processors in the array, P, is a parameter in the analysis.

The parallel loop program to be analyzed is shown in Figure 3. The call to function f in the loop body represents the loop body computation, and is a *neighborhood* computation. The output for such programs can be computed in parallel using both data partitioning and loop body pipelining techniques.

Two parameters of the program are required to analyze its execution when mapped onto the linear processor array. The first parameter is the loop body execution cost, which is represented by BB_i for each processor i. BB_i represents the number of instructions (or machine cycles) required on processor i to execute one iteration of the loop. The second program parameter is the size of the data set, N. To simplify the analysis, we only analyze parallel loop programs that have one-dimensional data sets as input. The execution models can be extended to data sets with more than one dimension [18]. A data set of size N implies that there are N total loop iterations to be performed across all the processors in the linear array.

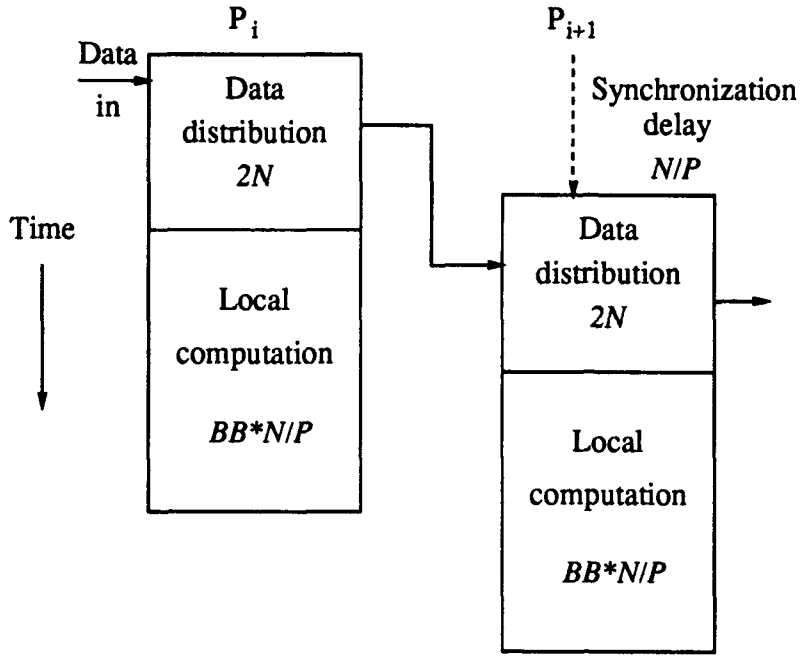


Figure 4: Execution time behavior for block data partitioning a parallel loop

3.1 Data partitioning

Data partitioning divides the input (and output) data sets across the processors in the linear array, so each processor can compute its output in parallel. As shown in Figure 4, there are two phases to the mapped program; first distribute the input data set across the processors and then perform the loop body computation on the data stored locally. Because the loop body computation in Figure 3 is a neighborhood computation, the input data set can be partitioned across the processors (with some overlap between neighboring cells, if either x or y from Figure 3 is non-zero).

The general execution model can now be used to analyze the execution time behavior of the parallel loop program, mapped by data partitioning onto a linear processor array. In analyzing the execution of the program, we assume that the mapping technique generates homogeneous, or SPMD (single program, multiple data), code. This means that each processor executes the same program text. For such code,

$$BB_i = BB, \ \forall 1 \leq i \leq P$$

meaning that the loop body execution costs are the same on all processors.

Data partitioning assigns the computation for N/P output data items (loop iterations) to each processor. If P does not divide N, the data set can be padded so that all processors are assigned the same number of data items ($\lceil N/P \rceil$). Because the data partitioning technique generates homogeneous code, the local computation term is the same for all processors ($SEQ_i = SEQ$). Therefore, the local

computation cost in the general execution model is

$$SEQ_i = SEQ = \lceil N/P \rceil \cdot BB$$

For the rest of the analysis, we will ignore the ceiling on N/P, to simplify the presentation of the modeled execution costs by allowing the combining of terms from the various components of the execution models.

For the communication term in the execution model, we assume that each processor receives and sends all N data items, so

$$CL_i = CL = 2N$$

Overlap between local computation and communication is not as straightforward to model. Since the mapping technique generates the same code for each processor, the overlap is the same on all processors ($OL_i = OL$). OL is modeled as a fraction of the communication cost, so

$$OL = K \cdot CL,\ 0 \leq K \leq 1$$

For example, the model for overlapping *all* communication with computation on processor i, except the receives and sends of the input data that will be stored on processor i (i.e. I/O for the fraction of the data set stored on processor i, $1/P$, is not overlapped with computation), is

$$OL_i = OL = (1 - 1/P) \cdot CL = \frac{P-1}{P} \cdot CL = \frac{P-1}{P} \cdot 2N$$

The last term in the general execution model is synchronization delay. For this analysis, assume that the input data are block partitioned, so the first processor is assigned the computation for the first N/P loop iterations, the second processor the second N/P loop iterations, etc. This assignment of the computation to processors naturally partitions the input and output data sets onto the processors in the linear array by blocks of data with consecutive indices. For block data partitioning,

$$SD_i = (i-1) \cdot N/P$$

because processor i cannot start receiving its local input data until after processor $i-1$ has completed receiving its local input data.

To complete the analysis, we must find the processor that takes the longest time to complete execution. The execution model terms for all processors are the same, except for SD_i. For SD_i, processor P waits the longest, with

$$SD_P = (P-1) \cdot N/P$$

Therefore, the execution cost for processor i, T_i, is greatest for processor P in the linear array, and with $T_i = SEQ_i + CL_i - OL_i + SD_i$, the total execution cost, T, is shown in Equation (3).

$$T = T_P = (N/P) \cdot BB + 2N - K \cdot 2N + (P-1) \cdot N/P = (BB + 3P - 2K \cdot P - 1) \cdot N/P \quad (3)$$

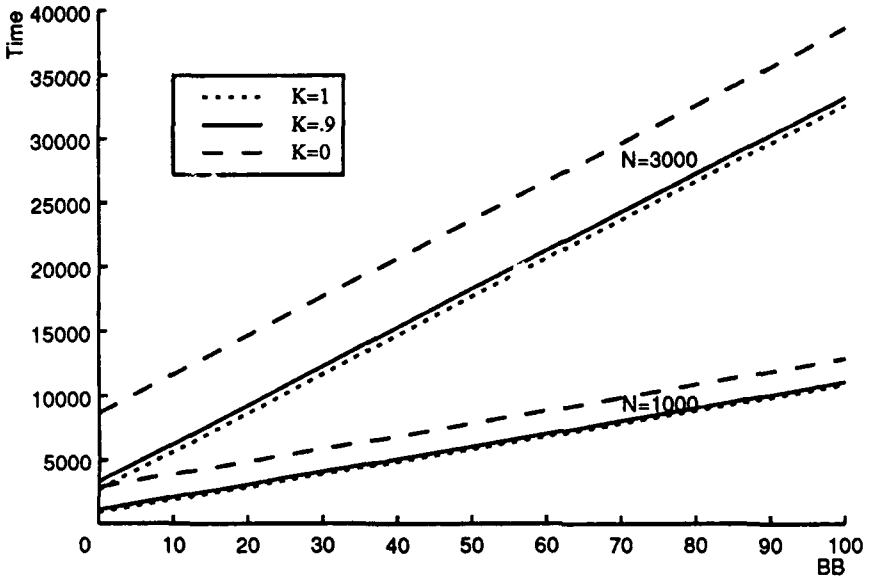


Figure 5: Varying overlap of communication and computation

Now that we have a general execution model for block data partitioning a parallel loop on a linear processor array, we investigate how sensitive the execution cost of the program, as modeled in Equation (3), is to a change in one of the parameters of the model. The analysis examines the effect of varying the overlap between communication and computation on the processors. Figure 5 plots program execution cost against loop body execution cost (BB), on a ten processor array ($P = 10$), for data set sizes of 1000 and 3000 ($N = 1000, 3000$). With $OL = K \cdot CL$, curves are shown for $K = 0$, corresponding to no overlap on each processor between communication and computation, $K = 1$, corresponding to complete overlap, and $K = .9$, for which communication for all data not stored on a processor (but only passed through to be stored on another processor) is overlapped with computation ($K = 1 - 1/P$).

Figure 5 shows that the amount of overlap between communication and computation can significantly affect the execution cost of the block data partitioned parallel loop. In particular, in varying the overlap, the relative differences in execution costs (slowdown factor relative to complete overlap) are larger for smaller N, but absolute execution cost differences are larger for larger N. For fixed data set size N, decreasing the fraction of overlap between communication and computation can cause a large, but constant, increase in program execution cost with increasing loop body execution cost. For example, for $N = 3000$, the program with no overlap between communication and computation will have an execution cost 6000 greater than the program that overlaps all communication and computation, for any loop body execution cost BB. For small BB, that difference is a significant fraction of total program execution cost.

Another form of data partitioning can be applied to the parallel loop program in Figure 3. For a P processor linear array, with both processor numbers and data indices starting from one, interleaved data partitioning assigns the first processor all data items with indices congruent to 1 mod P, the second processor all data items with indices congruent to 2 mod P, etc. Interleaved data partitioning is mainly useful for programs that do not require overlapping input data sets on neighboring processors (i.e. $x = 0$ and $y = 0$ in the parallel loop program from Figure 3), otherwise each data item would have to be replicated on several $(x+y+1)$ processors, leading to inefficient use of local memory on the processors.

As for block data partitioning, interleaved data partitioning assigns the computation for N/P loop iterations to each processor in the linear array. For interleaved data partitioning, the local computation (SEQ), communication (CL) and overlap (OL) terms in the general execution model are the same as for block data partitioning, because each processor is still doing the same computation and communication, but stores and computes a different part of the data set. However, the synchronization delay term for processor i is

$$SD_i = i - 1$$

because each processor can receive one data item and then pass the next $P-1$ data items immediately, so no processor has to wait long to start receiving its local data. For interleaved data partitioning, again with $OL = K \cdot CL$, $0 \leq K \leq 1$, processor P takes the longest time to complete execution (because of the SD_i term in the model). The execution cost for the complete program, T, is shown in Equation (4).

$$T = T_P = (N/P) \cdot BB + 2N - K \cdot 2N + P - 1 = (BB + 2P \cdot (1-K)) \cdot N/P + P - 1 \quad (4)$$

We can now compare the execution costs of the block data partitioning and interleaved data partitioning mapping techniques on the parallel loop program from Figure 3. Figure 6 plots the relationship between data set size N and the parallel loop program execution cost, for a ten processor linear array ($P = 10$), with a loop body execution cost BB of 10. With $OL = K \cdot CL$, the overlap between communication and computation on a processor varies from none ($K = 0$), to half ($K = .5$), to overlap of communication for all data except that of data that is stored on the cell ($K = .9$).

Figure 6 shows that the program, mapped with interleaved data partitioning, can perform significantly better than the same program, mapped by block data partitioning (by a constant factor), for fixed loop body execution cost (BB). The improved performance comes from the difference in the synchronization delay term in the execution model (SD_i), which is proportional to P for interleaved data partitioning, as opposed to being proportional to N for block data partitioning. The difference in execution cost between block data partitioning, T_{bdp}, and interleaved data partitioning, T_{idp}, is determined by the difference in synchronization costs, which is

$$T_{bdp} - T_{idp} = (P-1) \cdot N/P - (P-1) = (P-1) \cdot (N/P - 1)$$

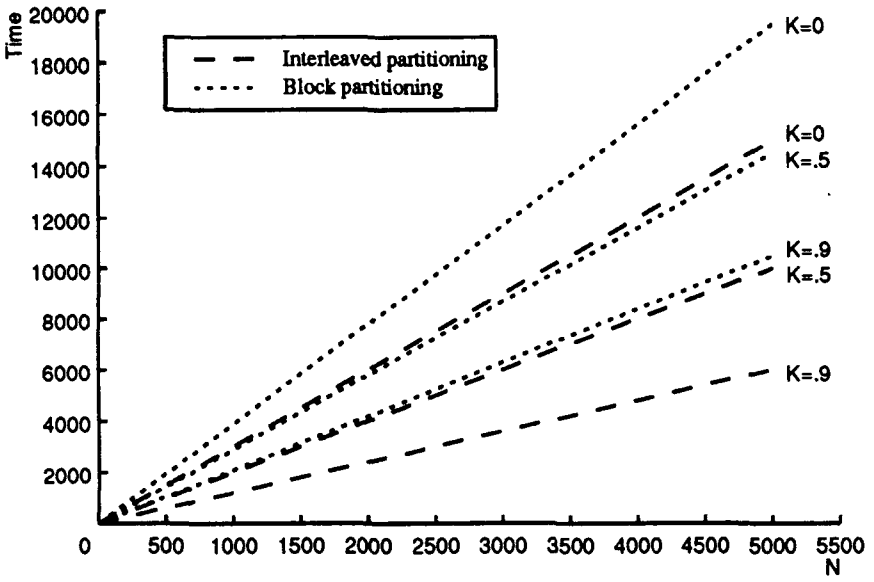


Figure 6: Block vs. interleaved data partitioning ($BB = 10$)

Since the data set size for a program (N) is almost always larger than the number of processors (P), the difference is positive, so interleaved data partitioning produces faster programs.

For the same overlap of communication and computation (K), interleaved data partitioning can be *much* better than block data partitioning. The difference in synchronization costs shown above is proportional to data set size (N), but is only a significant fraction of the total execution cost, for either block or interleaved data partitioning, for relatively small loop body execution cost (BB). For programs with small BB, as in Figure 6, interleaved data partitioning can provide up to a factor of two improvement in performance relative to block data partitioning. The greatest performance improvements from interleaved data partitioning appear in mapped programs with a high proportion of overlap between communication and communication (e.g. $K = .9$), since those are the programs with the smallest total execution costs.

The analysis of the interleaved and block data partitioning mapping techniques also shows that the synchronization delay term (SD_i) in the general execution model can be critical in selecting the best mapping technique from among those applicable to a particular program. In analyzing the performance of mapped programs, synchronization delay is often the most difficult component of the general execution model to determine, because it is a *global* property of the mapped program. All other terms in the general model can be determined by local analysis of the computation and communication that each processor must perform. However, synchronization delay depends on interactions between multiple processors, so must be determined by examining the execution behavior of the mapped program

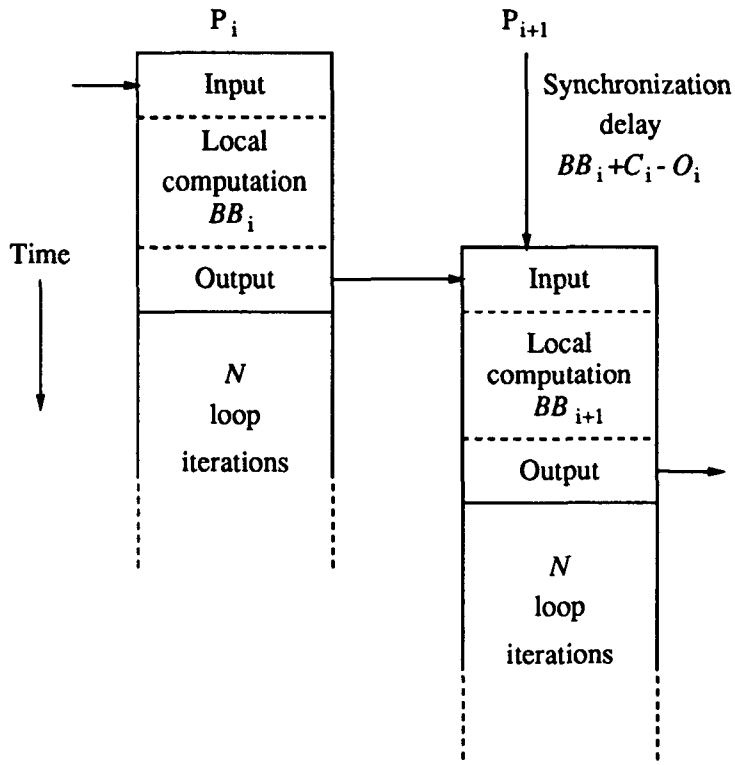


Figure 7: Execution time behavior for pipelining the body of a parallel loop

on *all* processors.

3.2 Loop body pipelining

Loop body pipelining assigns a part of the loop body computation of the parallel loop to each processor in the linear array [9]. The intermediate results for each loop iteration flow between processors, with multiple loop iterations executing in parallel once the pipeline fills. Loop body pipelining is similar to the DOPIPE loop transformation [19] that has been proposed for separating a loop body into multiple stages that can be assigned to distinct processors. As shown in Figure 7, the loop body pipelining mapping technique assigns all N loop iterations of the parallel loop program from Figure 3 to each processor, with each processor performing part of the loop body computation for each iteration.

The general execution model can be applied to the parallel loop program, mapped by loop body pipelining, onto the linear processor array. Loop body pipelining generates heterogeneous code for the processors in the linear array, meaning that each processor executes different program text. In the execution model, BB_i represents the execution cost of the loop body computation assigned to processor i. The local computation cost for processor i in the general execution

model is then given by

$$SEQ_i = N \cdot BB_i$$

For the communication term, C_i is the execution cost of communication for processor i in each loop iteration, and includes both receiving input from processor $i-1$ and sending output to processor $i+1$. Then, for N loop iterations,

$$CL_i = N \cdot C_i$$

Overlap between communication and local computation for one loop iteration is modeled as O_i, so the total overlap term is

$$OL_i = N \cdot O_i$$

To model the synchronization delay term, assume that the processor executes a loop iteration in three phases, as shown in Figure 7. The first phase receives input, the second phase performs the loop body computation and the third phase sends output. The synchronization delay is determined by the time to fill the pipeline, because a processor can start its local computation as soon as all the processors before it in the pipeline perform one loop iteration (including receiving input and sending output). There is no synchronization delay for the first processor in the linear array, and the synchronization delay for processors two through P can be expressed as

$$SD_i = SD_{i-1} + BB_{i-1} + C_{i-1} - O_{i-1} = \sum_{j=1}^{i-1}(BB_j + C_j - O_j)$$

With $T_i = SEQ_i + CL_i - OL_i + SD_i$, the total execution cost for processor i in the linear array is shown in Equation (5).

$$T_i = (BB_i + C_i - O_i) \cdot N + \sum_{j=1}^{i-1}(BB_j + C_j - O_j) \tag{5}$$

The last step is to find the processor that takes the longest to complete execution. To simplify the complete analysis, we assume that $\forall i$, $O_i = C_i$, meaning that *all* communication of intermediate results (and program input and output) is overlapped with the loop body computation on each processor. This assumption is not unrealistic, because only the communication for one iteration of the loop (of intermediate results) must be overlapped with the loop body computation assigned to a processor. With that assumption, the execution cost for each processor is given by Equation (6).

$$T_i = BB_i \cdot N + \sum_{j=1}^{i-1} BB_j \tag{6}$$

The analysis is presented in two parts; first for the computation load equally balanced across the processors (i.e. $\forall 1 \leq i,j \leq P,\ BB_i = BB_j$), and then for the computation load not balanced across the processors. For both parts, BB is the execution cost for the complete body of the parallel loop.

For the first part of the analysis, the computation load across the processors in the linear array is balanced, implying that

$$\forall 1 \leq i \leq P,\ BB_i = \lceil BB/P \rceil$$

In this case, processor P completes execution after all other processors, since it has the largest synchronization delay, which is the cost for the other $P-1$ processors to complete one loop iteration ($(P-1)\cdot(BB/P)$). Assuming that P divides BB, applying Equation (6) produces the execution cost given by Equation (7) for the loop body pipelined parallel loop.

$$T = T_P = (BB/P)\cdot N + (P-1)\cdot(BB/P) = (BB/P)\cdot(N+P-1) \quad (7)$$

In the second part of the analysis, with the computation load not balanced across the processors, there is some processor j assigned the most computation per loop iteration. In other words, the loop body computation assigned to processor j has the largest execution cost over all the loop body segments assigned to the processors. This is modeled as

$$BB_j = LF \cdot BB/P$$

and $1 < LF \leq P$, since the total execution cost of the complete loop body is BB. We call LF the *computation load factor*, because it determines the computation load on the processor that is performing the most work, which limits the performance of the mapped program. To simplify the application of Equation (6), assume that $j = P$, so the last processor in the linear array both performs the most computation per loop iteration and has the greatest synchronization delay. The synchronization delay is, therefore, $BB - LF \cdot BB/P$, since that is the cost for one iteration of the loop body passing through all processors, except the last processor. The execution cost for the parallel loop is determined by processor P and is given by Equation (8).

$$T = T_P = (LF\cdot BB/P)\cdot N + (BB - LF\cdot BB/P) = (BB/P)\cdot(LF\cdot N + P - LF) \quad (8)$$

Figure 8 shows the effect that the computation load factor, LF, has on the execution time of the parallel loop program, mapped by loop body pipelining. Figure 8 plots program execution cost vs. data set size (N) on a ten processor linear array ($P = 10$), for loop body execution costs (BB) of 10, 50 and 100.

The computation load factor, LF, is a multiplicative factor in the slope of the curves shown in Figure 8. In particular, Figure 8 shows that having the processor assigned the most computation per loop iteration (as measured by execution cost) perform twice the average per processor amount of computation per loop iteration

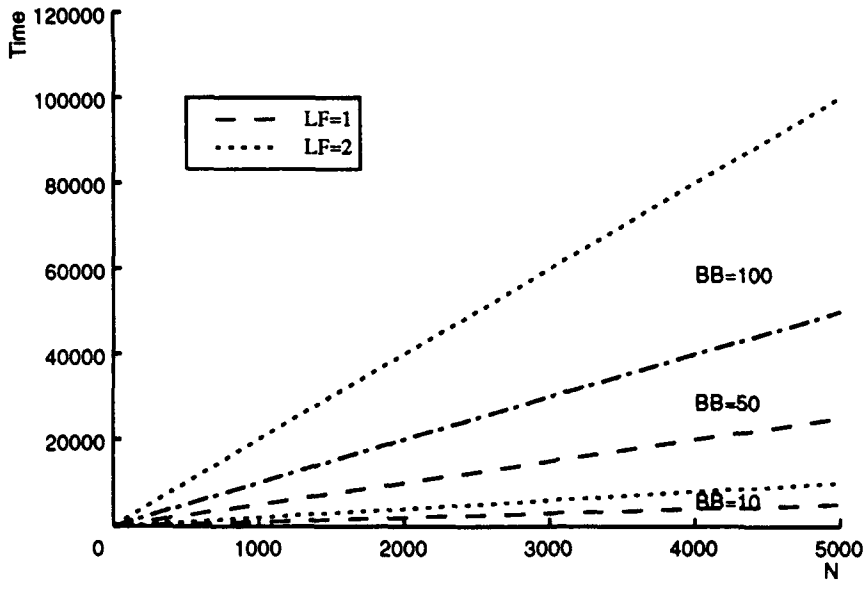


Figure 8: Varying BB, with computation load balanced (LF=*1*) and unbalanced (LF=*2*)

($LF = 2$) is equivalent to doubling the amount of computation in the entire loop body with optimal assignment of computation to processors ($LF = 1$) (i.e. so *all* the processors do twice as much computation per loop iteration). This effect is highlighted in Figure 8, in which the line for $K = 1$ with $BB = 50$ coincides with the line for $K = 2$ with $BB = 100$. This result implies that effective techniques are required for evenly partitioning the computation in the loop body onto the processors in the linear array, to even consider applying loop body pipelining to a parallel loop.

3.3 Comparative analysis

We have presented the execution models for two different techniques (data partitioning and loop body pipelining) for mapping the parallel loop program in Figure 3 onto a linear processor array. Because we are interested in mapping the program in the most efficient way onto the linear array, in this section we compare the two mapping techniques under a variety of values for the parameters of the program (N and BB). For the block data partitioning technique, we also vary the degree of overlap between communication and computation and, for loop body pipelining, we vary the computation load factor (LF). In this section, we only consider a ten processor linear array ($P = 10$).

The first analysis, shown in Figure 9, plots program execution cost against loop body execution cost (BB), fixing the data set size (N) at 1000. For block data partitioning, three curves are shown, varying the overlap of communication and

computation from no overlap ($OL = 0$), to overlapping half the communication with computation ($OL = .5 \cdot CL$), to complete overlap between communication and computation ($OL = CL$). For loop body pipelining, curves are shown with computation load factors (LF) of 1, 1.5 and 2.

Figure 9 shows that the execution cost of the block data partitioned program is not particularly sensitive to the overlap between communication and computation on each processor, as is shown by the small upward shift in the curves with decreasing overlap. On the other hand, the performance of the program, mapped by loop body pipelining, is *very* sensitive to the computation load factor (LF). An increase in the computation load factor increases the *slope* of the curve plotting program execution cost against loop body execution cost, rather than just shifting the curve. Overall, Figure 9 shows that block data partitioning usually performs at least as well as loop body pipelining. However, loop body pipelining can perform better than block data partitioning, but only with good load balancing across the processors in the linear array ($LF \approx 1$).

In Figure 10, with $N = 10,000$, the effects from Figure 9, with $N = 1000$, are exaggerated, so loop body pipelining with perfect load balancing ($LF = 1$) is better than block data partitioning by even more (about 9000 instead of 900). In general, the parallel loop program mapped by loop body pipelining can perform better than the block data partitioned program by an arbitrary amount (up to a factor of 2 for small BB and large N). Figure 10 also shows that, for block data partitioning, the overlap between communication and computation (OL) has a greater effect on program execution cost for larger data set sizes (N), especially for programs with relatively small loop body execution costs (BB). For large N (e.g. 10,000), loop body pipelining is better than block data partitioning, even with an unbalanced load ($LF > 1$), so long as BB is relatively small (e.g. for $OL = 0$ and $LF = 1.5$, loop body pipelining is better for $BB < 60$). For parallel loop programs with those characteristics, the overhead for distributing the data for data partitioning is a significant fraction of the total execution cost of the program, while the overhead for filling the pipeline for loop body pipelining does not depend on the data set size.

In general, the complete analysis shows that loop pipelining provides better performance for programs with small loop body execution cost (BB), as long as the work load is reasonably balanced ($LF \approx 1$). On the other hand, block data partitioning provides better performance for larger BB, so long as communication is overlapped with computation (OL is a large fraction of CL). In either case, the mapping technique that performs better can perform better by an arbitrary amount, meaning that selecting the better mapping technique, for a particular program instance, can be crucial in obtaining the best performance.

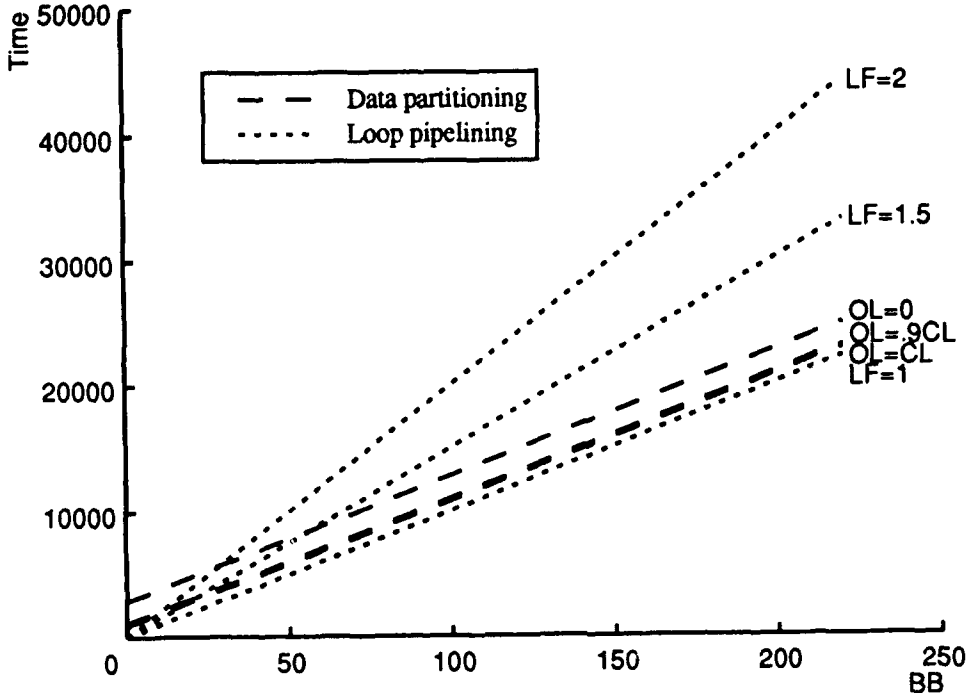


Figure 9: Block data partitioning vs. loop body pipelining, N=1000

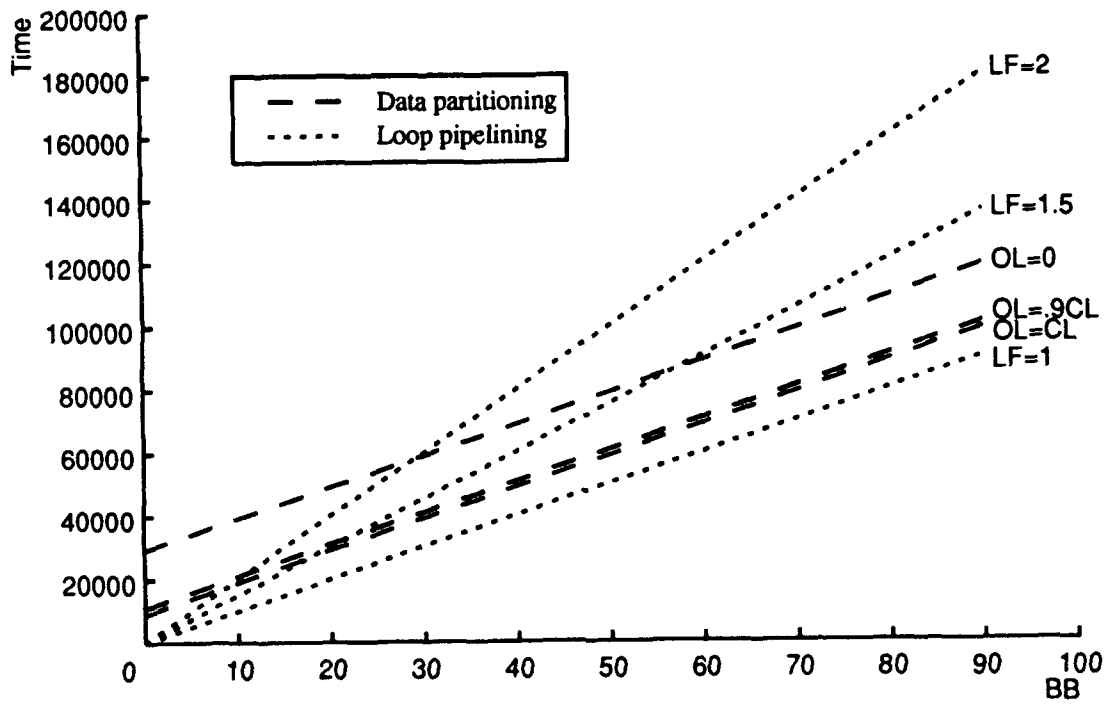


Figure 10: Block data partitioning vs. loop body pipelining, N=10,000

4 Experimental Results - Sisal on Warp

We have implemented a mapping compiler for the applicative programming language Sisal [12] on the Warp systolic array machine [2] to demonstrate that execution models can be used successfully to guide a mapping compiler for a distributed memory parallel processor. Sisal programs do *not* contain any provisions for interprocessor communication, so both the mapping onto processors and the generation of interprocessor communication must be done by the mapping compiler. The mapping compiler translates the IF1 dataflow graph representation of a Sisal program [17] into complete programs for the Warp machine.

The mapping compiler applies one or more mapping techniques to generate code for each processor in the Warp array. The compiler selects which mapping technique(s) to apply to a particular Sisal program automatically, using execution models developed for those mapping techniques on the Warp machine to select the technique(s) that generate the most efficient code from among the techniques implemented. The set of techniques implemented includes block and interleaved data partitioning, function body pipelining and forall loop body pipelining.

We present evidence that the execution models developed for the machine allow the compiler to select a good mapping technique for a particular program. If the execution models can accurately predict the performance of a program (or program segment) mapped in different ways, then the compiler can select the best method(s) for the program. For example, within the data partitioning mapping technique, choices must be made, such as whether to use block or interleaved data partitioning or whether it is worthwhile to overlap I/O and computation. The execution models for data partitioning predict that interleaved data partitioning is always better than block data partitioning, when it is reasonable to apply interleaved data partitioning. The models also predict that overlapping I/O and computation always is preferable to not doing the overlap. Both predictions are in accordance with the performance of programs generated using the various forms of data partitioning.

The major difficulty in applying the execution models relates to the complexity of the horizontal microcode of a Warp processor. On each clock cycle, a Warp processor may perform several operations, including integer and floating point arithmetic, local memory accesses and reads from and writes to the queues between neighboring processors. The cell compiler uses many complex techniques to optimize the use of the functional units in the processor. Therefore, instead of a single execution cost the mapping compiler uses both upper and lower bound estimates to characterize the execution time of a sequential block of cell code. The mapping compiler can then use those bounds to determine the best mapping strategy to use for a particular program (or program segment).

All of the benchmark programs can be mapped onto the Warp machine using more than one of the mapping techniques. The key result from executing the mapped benchmark programs on the systolic array machine is that the execution

models are accurate enough to always allow selection of the mapping technique that provides the best performance on the machine. For a given program instance, the models can predict which mapping technique performs best on the machine, but cannot always predict exactly how much better one mapping technique performs compared to another. Examples comparing the predictions of the execution models with the actual performance of programs on the Warp machine are presented.

4.1 Image processing

Two low-level image processing benchmarks illustrate the ability of the execution models to select the best mapping method from among the forms of data partitioning applicable to a program. Three sequential models, an upper bound model and two lower bound models (with and without software pipelining of innermost loops) have been applied to the image processing programs. All the models are able to order the techniques so the best one can be selected, and together the models are also able to bound the actual execution time of programs mapped onto the Warp machine. The lower bound sequential model that takes into account software pipelining is usually much too optimistic in predicting the execution time of mapped programs, so we concentrate on the upper bound model and lower bound model without software pipelining. Results are presented for both interleaved and block data partitioning, both with and without overlap of communication and computation. Interleaved data partitioning can be applied to the **addc1b** program because of the structure of the loop body computation, while the **asmt** program can only be block data partitioned.

Figure 11 shows both the actual times and the model predictions for the two image processing programs, presented to look at the question of how well the models order the applicable mapping techniques. Each vertical curve connects the points for a single mapping technique, across the actual times and estimates produced using all three sequential machine models (the horizontal lines). If the vertical curves cross, then either the various sequential models do not produce the same ordering among the different mapping techniques or the execution models do not correctly order the mapping techniques with respect to actual execution time on the machine. In fact the curves do *not* cross, so all the sequential models can be used to order the mapping techniques correctly for both of the image processing programs.

The other interesting question is how well do the sequential models for the Warp machine predict the execution time of data partitioned programs. Figure 12 shows the same data as in Figure 11, presented to show how well the sequential models bound the execution time of the mapped programs. In these graphs, we are looking to see whether the actual execution time of the programs is between the estimates using the lower and upper bound sequential machine models.

In Figure 12 the data appears somewhat inconclusive as to how good the estimates are at predicting actual execution times of data partitioned programs. For

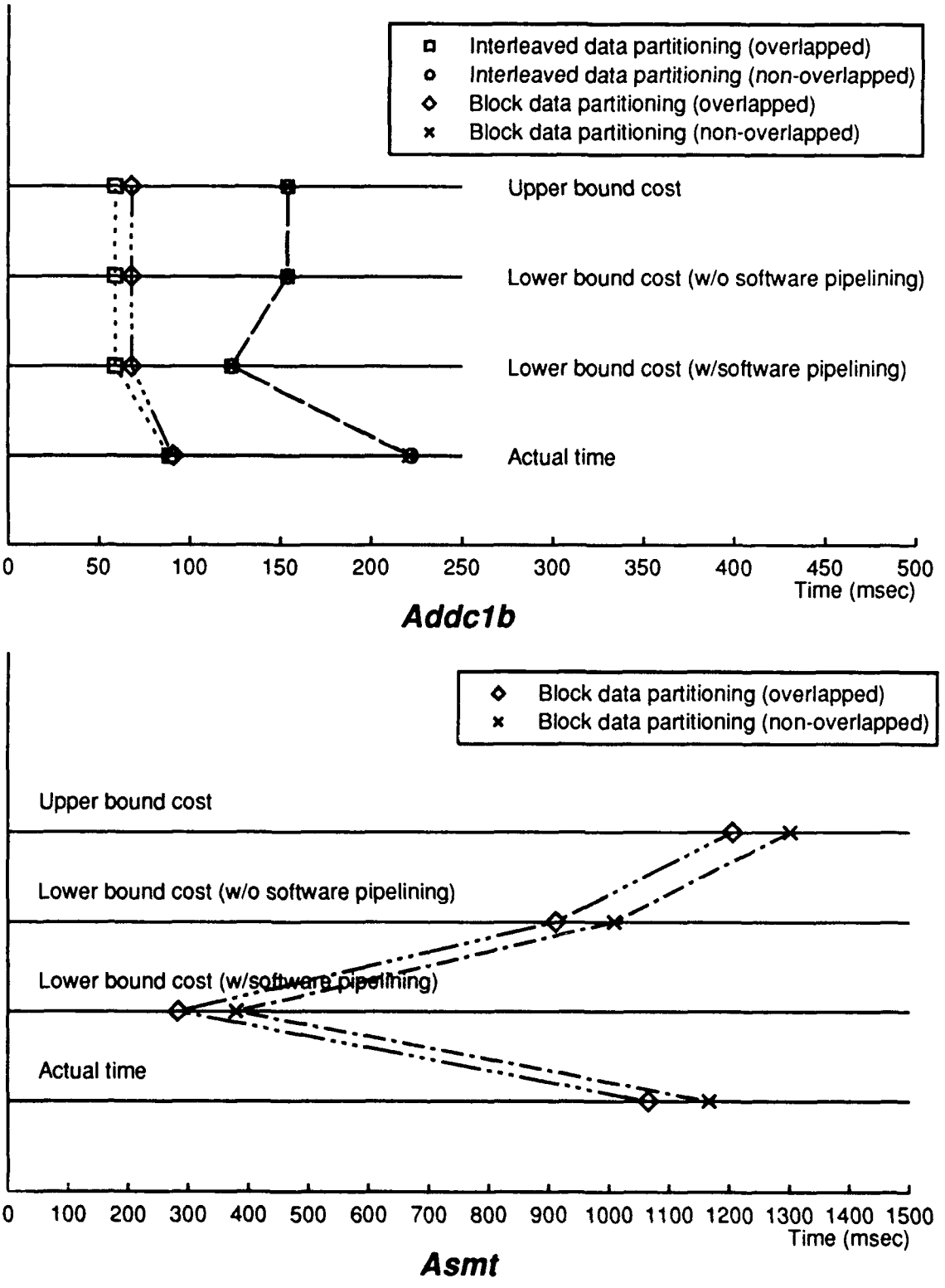


Figure 11: Image processing benchmarks, for ordering mapping techniques

addc1b, the upper bound estimate appears to be too low, because the program execution time is greater than the supposed upper bound estimate. However, note that there are *two* actual execution times for **addc1b**, for non-overlapped block and interleaved data partitioning. The larger time was generated under the same machine and compiler conditions as the other image processing benchmark time. The smaller time, which is much closer to the predicted time, was generated after fixing a bug in the W2 cell compiler for the Warp machine. The performance of **addc1b** is limited by the I/O capabilities of the Warp host, because of the small amount of computation in the loop body of the program. The bug in the cell compiler slowed down the host I/O significantly, so the program time was much higher than it should have been. In addition, the execution models do not take into account the capabilities of the host machine, only looking at the processors in the Warp array, so the true performance of any program that is limited by the host machine is not easily predictable.

On the other hand, for **asmt**, the models do a good job of bounding the execution time of the program mapped by block data partitioning.

In general, the models predict that, if applicable, interleaved data partitioning is at least as good as (and usually better than) block partitioning on a systolic array machine, and the actual execution times of the data partitioned image processing programs follow the predictions. Also, the models predict that overlapping communication and computation on the processors in the Warp machine is always better than not doing so, and the image processing programs support that prediction.

4.2 Polynomial evaluation

The polynomial evaluation benchmarks illustrate both the benefits of overlapping communication and computation for interleaved data partitioning and the relative merits of data partitioning and loop body pipelining for one type of program. Polynomials of degree 3, 5, 7, 9, 10, 12, 14 and 17 are used to evaluate the execution models. For these programs, the execution cost estimates for the loop bodies are the same whether using the upper bound sequential model or the lower bound model (without software pipelining), because the loop body consists of a single chain of data dependent operations (i.e. there is no parallelism in the operations for one loop iteration). Figure 13 shows the actual execution times of the polynomial evaluation programs mapped by data partitioning (both with and without overlap of communication and computation) and loop body pipelining. Figure 14 shows the predictions of the models using the upper bound sequential model and the lower bound model

For all polynomial degrees tested, the data partitioned program with overlap of communication and computation is predicted to perform better than the non-overlapped program, and the prediction holds up for the actual execution times. However, for polynomials of up to degree 10, the predictions are much too pessimistic for overlapped data partitioning, because of software pipelining of inner-

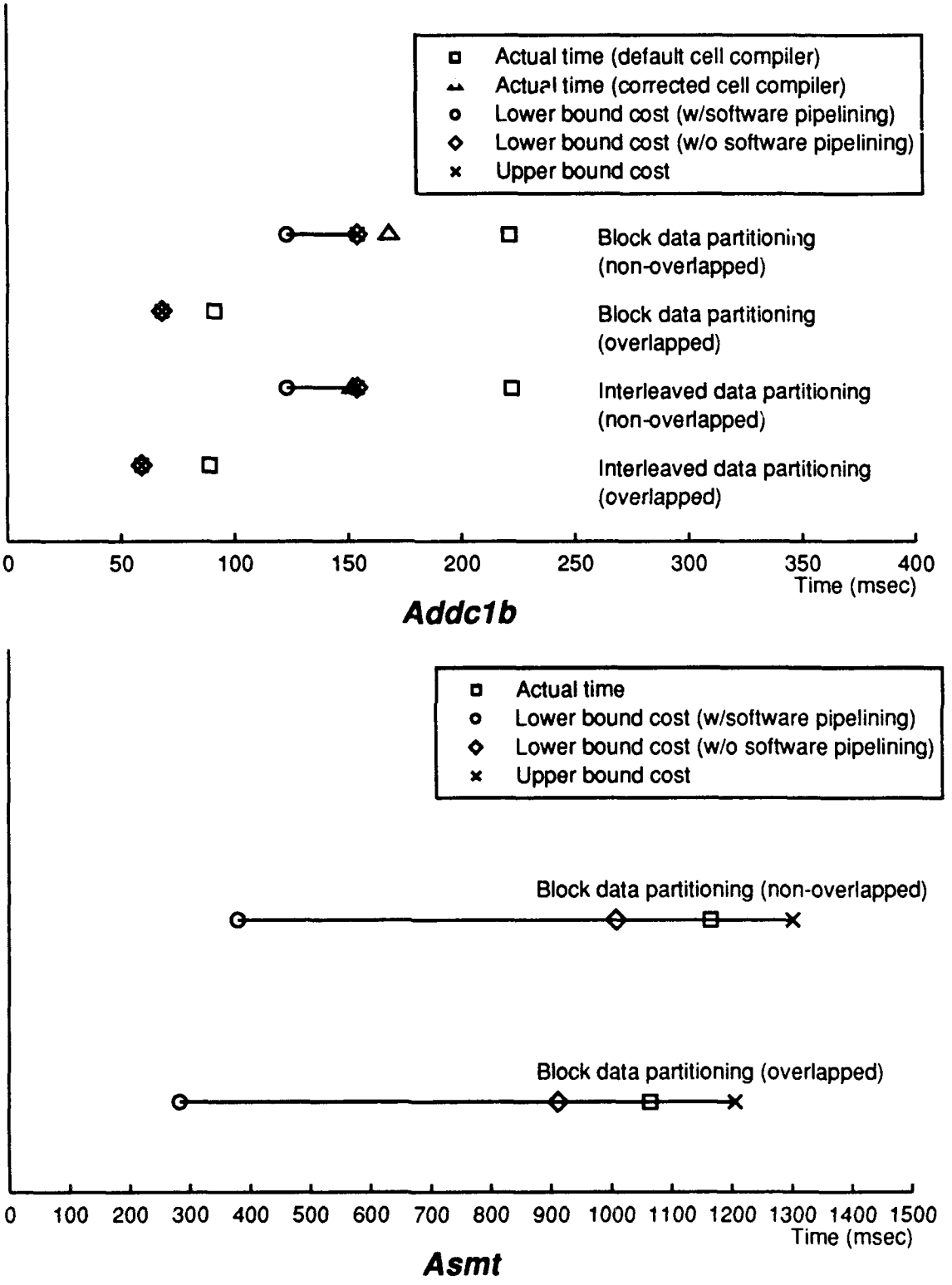


Figure 12: Image processing benchmarks, for predicting execution times

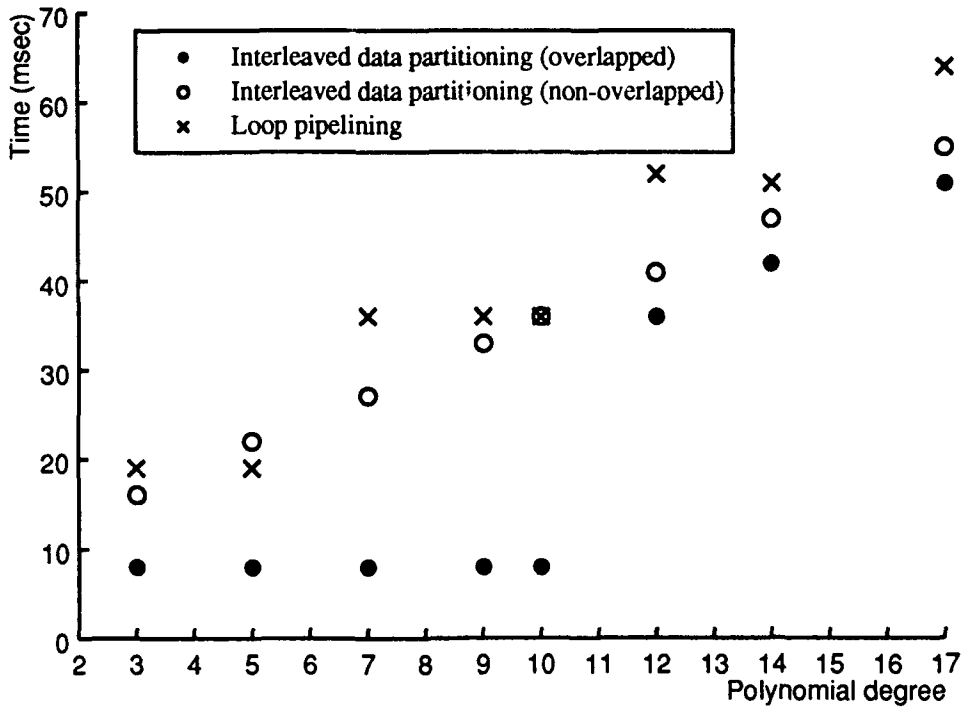


Figure 13: Polynomial evaluation, actual times

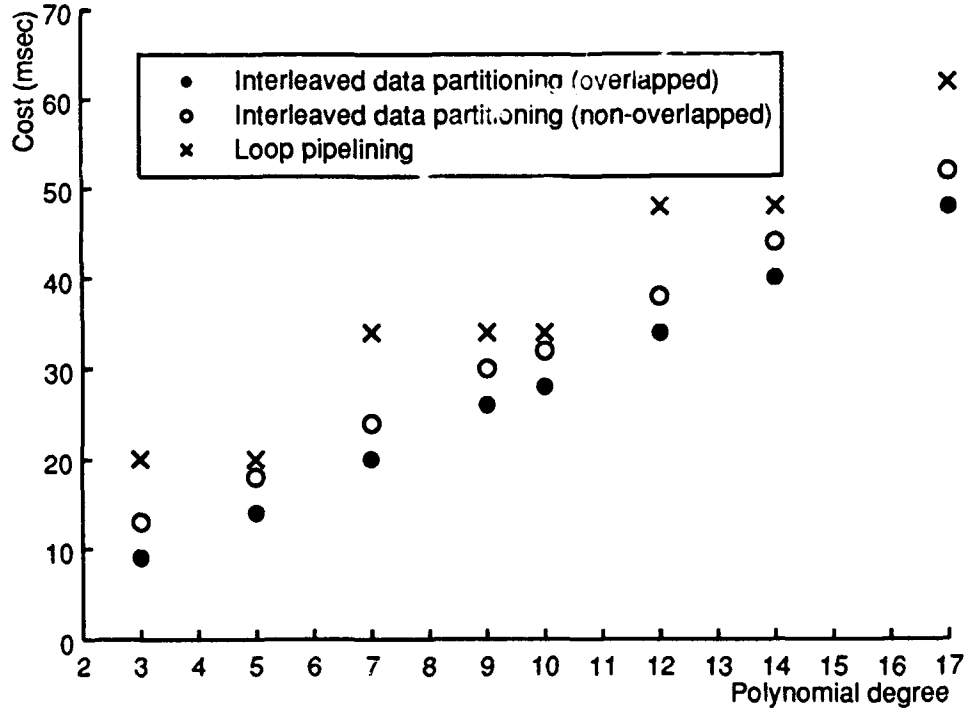


Figure 14: Polynomial evaluation, execution cost estimates

most loops. For those programs, a lower bound sequential model that includes software pipelining provides a more accurate estimate of actual execution time [18].

Loop body pipelining is also predicted to be inferior to overlapped data partitioning, and the prediction holds true on the machine. The only anomaly is for the degree 5 polynomial, for which the prediction says that loop body pipelining is slightly slower than non-overlapped interleaved data partitioning (20 vs. 18 milliseconds), while the actual performance of the loop pipelined program is somewhat better than the data partitioned version (19 vs. 22 msec). Since the difference in performance between the two versions of the mapped program is so small, for both the predicted and actual execution times, it is not surprising that the execution models cannot perfectly distinguish between them. This observation also applies to the degree 10 polynomial, for which loop body pipelining and non-overlapped data partitioning have the same execution times (36 msec).

5 Conclusions

In this paper, we have shown that the selection of the best mapping technique for a program can have a significant impact on performance, and that accurate execution models can be developed for a real distributed memory parallel machine. In using the general model of execution to analyze mapping techniques for a parallel loop program on a linear processor array, we have demonstrated that different instances of a single program can require different mapping techniques to obtain the best performance from the parallel machine. The results from the implementation of mapping techniques and execution models for the Warp systolic array machine show that the models can predict execution time accurately enough that a compiler can select the best technique from among those applicable to a particular program. Taken together, these two results allow us to conclude that execution models can be used effectively to drive the process of automatically and efficiently mapping programs onto a distributed memory parallel machine.

References

[1] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5(5):617–640, October 1988.

[2] Marco Annaratone, Emmanuel Arnould, Thomas Gross, H.T. Kung, Monica Lam, Onat Menzilcioglu, and Jon A. Webb. The Warp computer: Architecture, implementation, and performance. *IEEE Transactions on Computers*, C-36(12):1523–1538, December 1987.

[3] Daya Atapattu and Dennis Gannon. Building analytical models into an interactive performance prediction tool. In *Proceedings Supercomputing '89*, pages 521–530. ACM Press, November 1989.

[4] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 213–223. ACM Press, April 1991.

[5] Michael Burke, Ron Cytron, Jeanne Ferrante, Wilson Hsieh, Vivek Sarkar, and David Shields. Automatic discovery of parallelism: A tool and an experiment (extended abstract). In *Proceedings of the ACM/SIGPLAN PPEALS 1988*, pages 77–84. ACM Press, July 1988.

[6] Marina Chen, Young il Choo, and Jingke Li. Compiling parallel programs by optimizing performance. *Journal of Supercomputing*, 2(2):171–207, October 1988.

[7] Ron Cytron. Useful parallelism in a multiprocessing environment. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 450–457. IEEE Computer Society Press, August 1985.

[8] Geoffrey C. Fox. Parallel computing comes of age: supercomputer level parallel computations at Caltech. *Concurrency: Practice and Experience*, 1(1):63–103, September 1989.

[9] Thomas Gross and Alan Sussman. Mapping a single-assignment language onto the Warp systolic array. In Gilles Kahn, editor, *Proceedings of the 3rd Conference on Functional Programming Languages and Computer Architecture*, pages 347–363. Springer-Verlag, September 1987.

[10] David E. Hudak and Santosh G. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Conference Proceedings of the 1990 International Conference on Supercomputing*, pages 187–200. ACM Press, June 1990.

[11] C.-T. King, W.-H. Chou, and L.M. Ni. Pipelined data-parallel algorithms: Part I - concept and modeling. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):470–485, October 1990.

[12] James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirkham, Bill Noyce, and Robert Thomas. *SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual Version 1.2.* Lawrence Livermore National Laboratory, March 1985.

[13] Constantine D. Polychronopoulos, Milind Girkar, Mohammad Reza Haghighat, Chia Ling Lee, Bruce Leung, and Dale Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages II-39 – II-48. Pennsylvania State University Press, August 1989.

[14] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* PhD thesis, Computer Systems Laboratory, Stanford University, April 1987.

[15] Vivek Sarkar and John Hennessy. Compile-time partitioning and scheduling of parallel programs. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 17–26. ACM, June 1986.

[16] Vivek Sarkar and John Hennessy. Partitioning parallel programs for macro-dataflow. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 202–211. ACM, August 1986.

[17] Stephen Skedzielewski and John Glauert. *IF1: An Intermediate Form for Applicative Languages.* Lawrence Livermore National Laboratory, July 1985.

[18] Alan Sussman. *Model-Driven Mapping of Computation onto Distributed Memory Parallel Computers.* PhD thesis, Carnegie Mellon University, September 1991. Also available as Technical Report CMU-CS-91-187.

[19] Youfeng Wu and Ted G. Lewis. Parallelizing while loops. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II-1 – II-8. Pennsylvania State University Press, August 1990.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1992	Contractor Report

4. TITLE AND SUBTITLE
EXECUTION MODELS FOR MAPPING PROGRAMS ONTO DISTRIBUTED MEMORY PARALLEL COMPUTERS

5. FUNDING NUMBERS
C NAS1-18605
WU 505-90-52-01

6. AUTHOR(S)
Alan Sussman

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA 23665-5225

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 92-8

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-189613
ICASE Report No. 92-8

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Michael F. Card
Final Report

Submitted to the 20th International Conference on Parallel Processing, August 1992

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited

Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This paper addresses the problem of exploiting the parallelism available in a program to efficiently employ the resources of the target machine, in the context of building a mapping compiler for a distributed memory parallel machine. The paper describes using execution models to drive the process of mapping a program in the most efficient way onto a particular machine.

Through analysis of the execution models for several mapping techniques for one class of programs, we show that the selection of the best technique for a particular program instance can make a significant difference in performance. On the other hand, the results of benchmarks from an implementation of a mapping compiler show that our execution models are accurate enough to select the best mapping technique for a given program.

14. SUBJECT TERMS
performance modeling; modeling; compilers; systolic arrays

15. NUMBER OF PAGES
30

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		

NSN 7540-01-280-5500

Standard Form 298 (Rev 2-89)
Prescribed by ANSI Std 239-18
298-102